


**Published chapter**
Austin, J. and Filer, R. (1996) *Using correlation matrix memories for inferencing in expert systems.* In: Taylor, J.G. (ed), Neural Networks and their Applications. John Wiley and Sons Ltd., Chichester, UK, 229-244. ISBN 0471962821

# Using Correlation Matrix Memories for Inferencing in Expert Systems

J. Austin, R. Filer

## 16.1 Introduction

Rule-based reasoning has been the subject of a great deal of work in AI, and the work has resulted in a number of expert systems. Some expert systems have proved very useful, *e.g.* PROSPECTOR [7] and DENDRAL [10], but it is clear that the usefulness of an expert system is not necessarily the result of a particular architecture. Usefulness is much more likely to be related to the ability of an expert system to access relevant information, indeed a system that appears intelligent may simply be one that is able to access a great deal of information that is relevant to solving a particular problem. Although systems like PROSPECTOR and DENDRAL *are useful*, the nature of real world problems is such that systems tend to be brittle (brittleness is an inability to deal with partial or uncertain information, or to generalize). Despite having a knowledge base representing *15 person-years work*, INTERNIST-I [11], an expert system for medical diagnosis, was "unable to synthesize a general overview in complicated multi-system disorders" (by the authors' own admission).

Touretsky and Hinton were the first to emulate a symbolic, rule-based system in a connectionist architecture [18]. A connectionist approach held the promise of better performance with partial information and being gen-

erally less brittle. Whether or not this is the case, Touretsky and Hinton usefully demonstrated that connectionist networks are capable of symbolic reasoning. The systems due to Lange and Dyer (ROBIN: [9]) and Shastri and Ajjanagadde (SHRUTI: [15]) came later, and have knowledge bases more reminiscent of the real world. ROBIN uses "signatures", while SHRUTI relies on a more elegant, "temporal synchrony" to propagate variable bindings. These later models can loosely be described as "connectionist", but both are highly constrained networks. In both systems, knowledge is basically hand-encoded and no learning is possible. Knowledge is not distributed in any sense, in either model, which means that properties that might otherwise result from a distributed representation are lost (*e.g.* an ability to deal with partial information). Implementation in the software or hardware of a conventional computer would also be difficult.

Sun [17] devised a dual representational scheme, with both localist representation (of concepts) and distributed representation of what amount to sub-concepts. The localist level also uses a fuzzy evidential logic. The system is consequently better able to deal with partial information and in-exact matching. What is still needed is a connectionist solution that maintains a truly distributed knowledge representation. This chapter describes Correlation Matrix Memory (CMM) and the use of CMM as an inference engine [2]. This chapter is concerned with particular aspects of using CMM in an expert system, and shows that CMM is a valuable tool with some very useful properties.

### Outline of The Chapter

Section 16.2 describes CMM and the Dynamic Variable Binding Problem. Section 16.3 deals with how CMM is used as part of an inferencing engine [2]. Section 16.4 details the important performance characteristics of CMM.

## 16.2 Correlation Matrix Memory

CMM is a binary associative memory. For the purposes of our work, the remit of CMM is the fast, parallel matching of rules following predicate calculus, *e.g.*:

$$\begin{array}{llll} \texttt{IF} & \texttt{(A)} & \Rightarrow & \texttt{B} \\ \texttt{IF} & \texttt{(A} \wedge \texttt{B)} & \Rightarrow & \texttt{C} \\ \texttt{IF} & \texttt{(A} \vee \texttt{B)} & \Rightarrow & \texttt{C} \\ \texttt{IF} & \texttt{(NOT A)} & \Rightarrow & \texttt{D.} \end{array}$$

The system can deal with multiple arity rules (*i.e.* rules with multiple variables in the antecedent and consequent), with value inheritance and multiple occurrences of variables, and with the exclusive-OR problem. CMM is

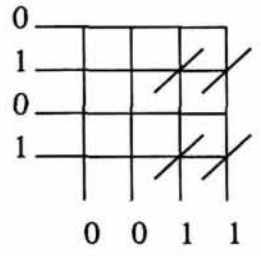


**Figure** 16.1: CMM Binary Associative Memory.

not a new idea (see [19]), which allows the association of binary vectors using a matrix of binary weights. Pairs of binary vectors are associated by setting weights as shown in Figure 16.1. As such, CMM can be seen as a single layer neural network with binary weights, which uses a Hebbian learning rule.

Figure 16.1 is an example of training, in which the vectors `0101` and `0011` have become associated. The subsequent presentation of `0101` to the network will retrieve `0011` if set weights in rows identified by set bits in the input vector contribute to column sums. The result is `0022`, which is then thresholded appropriately to give `0011`. For similar work, see Austin and Jackson [3]. Here, CMM is augmented by the use of Tensor Products (TP: [16]) to solve the dynamic variable binding problem.

## The Dynamic Variable Binding Problem

This is a problem in connectionist implementations of rule-based systems. It is best explained using an example: If a rule has the form $\mathtt{A} \wedge \mathtt{B} \Rightarrow \mathtt{C}$, it may be important to be able attribute values thus:

$$(\mathtt{A} = \mathtt{z}) \wedge (\mathtt{B} = \mathtt{k}) \Rightarrow \mathtt{C}.$$

When a distributed rule representation is to be used, it is important that both the $\mathtt{A}:\mathtt{z}$ and the $\mathtt{B}:\mathtt{k}$ bindings can be represented unambiguously. It is clearly useless if, having trained such a rule, the system is subsequently unable to "remember" which variable had the value $\mathtt{z}$. Furthermore, a rule may also specify inheritance:

$$(\mathtt{A} = \mathtt{z}) \wedge (\mathtt{B} = \mathtt{k}) \Rightarrow (\mathtt{C} = \mathtt{k}).$$

The binding representation used therefore has to be stable to propagation in the network. The problem of representing and propagating these bindings is what has become known as the "Dynamic Variable Binding Problem".

## 16.3 CMM Inference Engine

The system, proposed by Austin [2], consists of units that fall into two sub-categories:

1. CMM units (associative memory units);
2. Support units (not associative memory).

The support units perform the relatively simple processing necessary to support the CMM units, and exploit the technology to the full. This simple processing is all that is necessary to achieve a powerful reasoning capability.

Figure 16.2 illustrates the system. Processing occurs both at the input to the CMM units and at the output; processing fulfils the following functions, a description of which will serve to introduce the system.

### 16.3.1 Lexical Token Converter

Each lexical input item is converted to a binary token for manipulation by the system. Tokens are generated that consist of randomly allocated patterns of $N$ set bits out of a total of $M$ bits (there is an optimal ratio of $N : M$ that gives best error rate *vs.* storage). Random patterns may be allocated easily using a random number generator. Each token should be unique, however, which means a method of ensuring uniqueness is required. For few tokens it may be feasible to check a list of tokens each time a token is allocated; for many tokens there are better methods available, like "Test-and-Train" [4]. This method involves using the system itself to identify whether a pattern is already known. The values of $M$ and $N$ depend upon the overall size of the CMM units and the usual arity of rules being stored (see Section 16.3.3).

### 16.3.2 Binding Variable and Value Tokens

Binary tensor products (TP, [16]) are used to bind variables to values, TP vectors being obtained from a pair of tokens in two stages. The first stage is in fact analogous to storing a pair of tokens in a binary associative memory: with reference to Figure 16.1, the TP of `0101` and `0011` *is* the matrix of binary weights. The TP *vector* is obtained by concatenating rows, hence:

```
0000 0011 0000 0011
```

This is a binary vector of length $M^2$ containing $N^2$ set bits. It begs the question, whether to allow both tokens and TP vectors as representations in the CMM units. Clearly, if $N$ is chosen such that $N^2$ is optimal for error rate *vs.* storage in the CMM unit, $N$ itself cannot at the same time be optimal (see also Section 16.3.3). This is one reason why allowing both representations may be disadvantageous, which leaves us with the problem of what to do with

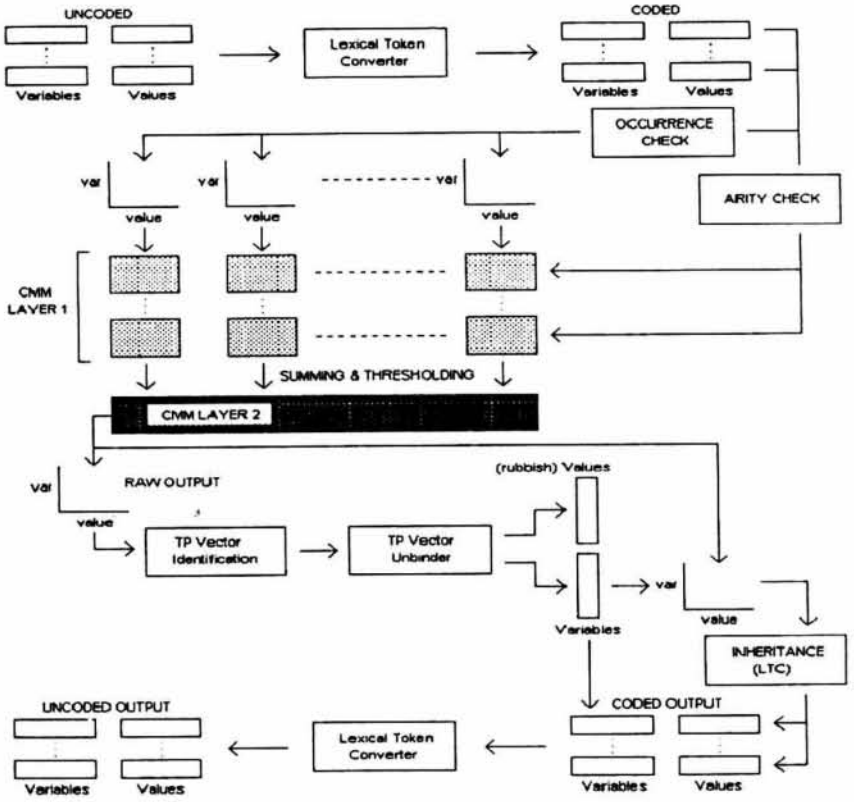


**Figure** 16.2: The CMM Inference Engine.

variables not assigned a value. Such variables can be assigned either a "null" value or a "true" value to allow conversion to TP vector form.

### 16.3.3 Superimposing Inputs

The TP vector(s) are superimposed (OR-ed on top of one another) prior to being applied to the CMM units. This makes dealing with commutativity of rule antecedents an *automatic feature of the system*, and is a key factor in providing an efficient partial match. Superimposing $k$ vectors with $N^2$ set bits gives rise to a vector with up to $kN^2$ set bits. It might therefore be more appropriate to optimize $sN^2$ for error rate *vs.* storage rather than $N^2$, where $s$ is chosen depending on the predominant arity of rules in the knowledge base.

### 16.3.4 Identifying CMM Units of Appropriate Arity

This is necessary to enable the appropriate processing of rules with mixed arity antecedents. For example, if the system has learned these two rules:

1. $\mathtt{A} \Rightarrow \mathtt{D}$.
2. $\mathtt{A} \wedge \mathtt{B} \Rightarrow \mathtt{E}$.

If the antecedents were trained in a single CMM unit, subsequent application of the token for A and thresholding appropriate for a single arity rule would allow *both* rules to fire (incorrectly). To avoid this, the first layer of CMM units must consist of a CMM unit *for each arity rule that will be encountered.* Input can then be targeted to the appropriate arity CMM unit.

### 16.3.5 Occurrence Checking

Very simple rules, such as the example in Section 16.3.6 (see Section 16.3.6), may involve repeated variables. For the correct processing of such rules, it is insufficient for the input simply to be superimposed and sent to the appropriate arity CMM unit, because the fact that a variable is repeated cannot be represented in this way. The solution is to extend further the first CMM layer, such that each arity CMM unit is duplicated, or triplicated even, depending on the application and the likely number of occurrences of the same variable in rules. This allows multiple occurrences of the same variable to be represented and, if summed appropriately, to count in thresholding.

### 16.3.6 Providing Separator Tokens (Training)

A single layer neural network cannot resolve the exclusive-OR problem. To do so requires a *two layer* network, and this approach has also been used in our system. For example, if separator tokens are represented here by {i,j,k,l}:

$$0 \wedge 0 \Rightarrow \mathtt{i} \Rightarrow 0$$
$$0 \wedge 1 \Rightarrow \mathtt{j} \Rightarrow 1$$

$$1 \wedge 1 \Rightarrow \quad \mathtt{k} \quad \Rightarrow 0$$
$$1 \wedge 0 \Rightarrow \quad \mathtt{1} \quad \Rightarrow 1$$

The support units provide a unique, randomly allocated separator token for each rule (the generation of tokens may be done off-line). The separator token is an $M^2$-bit word with a pattern of set bits, the number of set bits being optimized as before.

### 16.3.7 Thresholding (Retrieval)

Two sorts of thresholding are used: L-max [1, 5] and Willshaw [19]. L-max thresholding selects a *specified number of columns* in the output to represent set bits, columns being selected in order of decreasing sum magnitude; Willshaw thresholding takes *any column sums in the output above a specified value* as being set bits. It can be useful to apply both thresholds simultaneously.

### 16.3.8 Decoding the Output

The output of the system consists of all rule consequent(s) which match the given input, in the form of overlapped TPs. Decoding the output involves identifying explicitly which *tokens* may be present in the output in TP vector form. It is then possible to extract these TP(s) and use each token that has been identified to retrieve the other half of the pair of tokens represented in each TP. To this end, the TP is used itself like a binary associative memory (input retrieving output). With few rules matching, it is usually a simple task to identify unambiguously each component of the output. The more complex the output is, however, the more likely it is that spurious identifications are made. It is conceivable that a network approach will ultimately be used to decode the output. However, as this step of the processing is quite straightforward, the nature of the implementation is not crucial. An example of an ambiguous output is:

```
00000101001100000110001101001100000001101
```

Assume the TP vectors:

```
A  10000000010000011000000010000000000000100
B  00000101000000000010000001000000000001100
C  00000000000100000110000100000100000000001
D  10000010000000000100000000010000000000101
E  00000001100000000100000100000001000000010
F  00000000001100000000000100100110000000000
```

It can be seen that the 1s present in {B,C,F} are represented in the output, whilst the patterns {A,D,E} contain 1s that do not correspond to the output.

The system would deduce that the thresholded output is due to three rules matching. Suppose that this output was obtained after training only two rules, however. One of the identifications would therefore have been spurious.

## 16.4 System Characteristics

The next section describes how the CMM compares with a conventional storage method when used in a rule matching context. Some new work on storage and partial matching is presented (taken from Filer, [6]).

### 16.4.1 Storage

The storage of sparsely coded associative networks was investigated by Nadal and Toulouse [12]. In this paper, Information Theory was applied to the problem of predicting the maximum number of associations (TP vector pairs) that can be trained before a certain error rate is reached. The error rate is expressed in terms of the likely number of bits that are retrieved in error following a Willshaw threshold. With a single layer, square, binary associative network, Nadal and Toulouse showed that the maximum number of associations, for an error rate of 1.0 bits per retrieval, can be expressed in terms of the dimensions of the network alone:

$$\text{storage}_{1.0} = (\ln 2)^3 M^2/(\ln M)^2$$

It is also possible to predict storage for specific error rates ([6]; see graph, Figure 16.3), and the results are encouraging. For instance, if an error rate of one erroneous set bit per output is accepted (which is the same as saying a 100% 1-bit error rate), a 1000bit × 1000bit CMM can store 7000 associations[1]. With a 50% 1-bit error rate, 86% of this storage can still be used (6000 associations); with a 10% 1-bit error rate, 63% of the storage can still be used. Perhaps the most important point is that *storage in the network is greater than would be expected of a list storage system using the same amount of memory.*

The storage obtained is a compromise between the number of associations stored and the error rate, and depends on a certain coding rate, with optimal storage arising when there are $\log_2 M$ set bits out of $M$ bits. A lower coding rate gives rise to a greater error rate (but greater storage), whilst a higher coding rate leads to reduced storage (but less errors). In the current system, which supports multi-arity rules, the optimal value of $N$ is required (see also Section 16.3.3). If there is a choice, the largest $N$ should be chosen, as it is better to sacrifice storage than to incur a greater error rate.

[1] An optimal coding rate is assumed here.

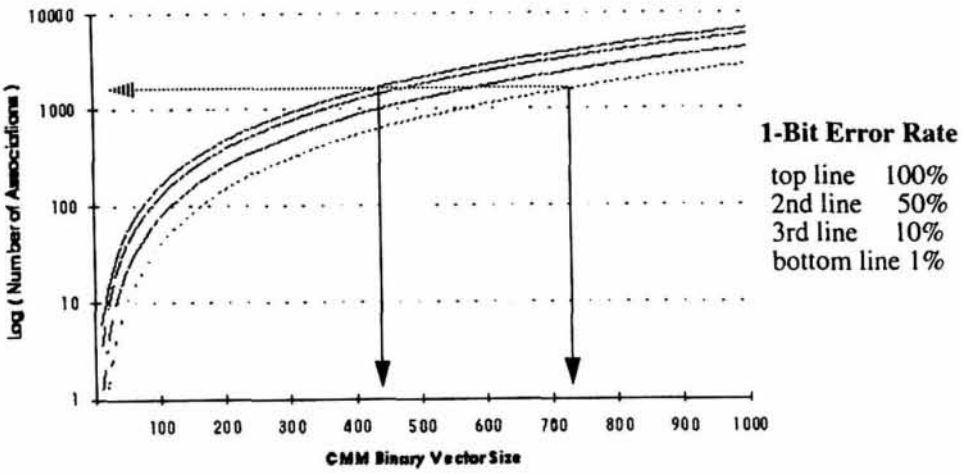


**Figure** 16.3: Storage capacity *vs.* CMM size ($M$) for various 1-bit error rates. This graph shows the fact that $M$ need increase by only a small amount, to lessen the error rate dramatically.

### 16.4.2 Partial Matching Capability

CMM can perform partial matching. The ability to perform inferencing based on partial information is a significant feature of human intelligence and, in many applications, the usefulness of an expert system might depend upon emulating this ability. Partial match is already implemented in standard database systems, and so a comparison between CMM and a database access method is possibly valid. In particular, the Superimposed Coding method (SIC; [8]), which is used in databases for partial matching, is very similar to the methods used here. Sacks-Davis and Ramamohanarao (S&R; [14]) suggested a superimposed coding scheme, called "Two Level Superimposed Coding", which I will now describe.

In brief, SIC is a way of indexing a store of records, which may be held on a slow storage device, where each record is given a code, a "record descriptor code", computed based on the particular attribute values of each record. Every possible attribute-value instance is allocated a unique binary vector token. Each record descriptor code is generated by superimposing all the tokens which are appropriate for that record. A query descriptor code can be computed in an analogous manner from one or more specified attribute values, and can then be used to search the file of record descriptor codes. If a bit

that is set in the query descriptor code is also set in a record descriptor code, then it is likely that there is a match. The system accept a false positive rate, which is also a feature of CMM. Importantly, however, no *true* matches are missed (which would clearly be unacceptable in a database retrieval system).

Two Level SIC is an improvement (rather like two level memory hierarchies in conventional computer systems), whereby a primary file and a secondary file are stored. The additional file stores so-called "segment descriptor codes". The file of record descriptor codes is subdivided into segments, and each segment is characterized by a segment descriptor code. Each segment descriptor code is computed from the records represented by that segment, but using a different hash function. The segment descriptor codes are then used to narrow down the search of record descriptor codes. Depending on the choice of segment size and how well similar record descriptor codes can be grouped together in the same segment, Two Level SIC performs well. Even in the worst case, which would be when all segments have to be examined, this scheme performs no worse than the one level scheme described by Knuth [8]. Usually with Two Level SIC though, the number of disk accesses (and hence speed of matching) is related more to the *actual number of records that match* than to the total number of records.

The above property is extremely desirable, and is not found with other techniques. Rivest [13] looked at several approaches: hash coding methods and tree search algorithms. Performance was considered in terms of retrieving $nk$-bit words that match a query specifying $s$ bits, and it was found that the best these approaches could offer was a performance= $O(n^{\log(2-s/k)})$. CMM is capable of more efficient partial match, similar to Two Level SIC. To demonstrate this fact, the remainder of this subsection is devoted to a consideration of the theoretical performance of CMM. CMM can be used to associate a SIC-type coding of attribute values with a separator code fulfilling a pointer function. This is analogous to the role of the first CMM layer in the inference engine already described.

#### 16.4.2.1 Attribute Value Coding

Input to the CMM would represent records to be found, with attribute values processed into a form of record descriptor code. An $M$-bit descriptor code should contain about $N$ set bits, where $N = \log_2 M$ for optimal performance (see Section 16.4.1). Consequently the mapping of attribute values should be onto an $M$-bit vector with $N/s$ set bits, where $s$ is the number of attributes in each record. This is analogous to the processing of inputs in the inference engine, omitting TP binding. A query descriptor code would be computed in an analogous manner to the descriptor code.

#### 16.4.2.2 Separator Coding

The output of the CMM may well contain overlapped separator codes. The separator codes must therefore be held in a separate structure, such that the individual separator codes represented can be identified in the output (as in Section 16.3.8). It is only necessary to store the $N$ numbers that describe the positions of the set bits, however, allowing a considerable memory saving. For the fastest possible access, a method is needed to process the output efficiently. To this end, the file of separator codes is subdivided and organized into buckets according to where the middle 1 appears in the code. To decode the output, it is then only necessary to download all codes that could be represented in the output, starting with the bucket corresponding to the position of the first 1 that could be the middle 1 of a separator code; the next area downloaded corresponds to the position of the second possible middle 1, *etc.*.

#### 16.4.2.3 Thresholding

It is appropriate to perform a Willshaw threshold [19], and at a level equal to the number of set bits in the query descriptor code. Such a threshold will not exclude false matches, but equally does not exclude any true matches (this being the desirable compromise). False matches arise from two sources: from false matches due to error bits; also from false matches due to spurious separator codes in the output. The number of false matches therefore depends upon the saturation of the memory and on the number of set bits in the query descriptor code.

#### 16.4.2.4 Theoretical Performance

We now calculate the amount of data that would need to be retrieved from a secondary store, directly related to the speed of the proposed partial matching system. There are two components:

1. The amount of data needed to compute the CMM output (*i.e.* that constitutes the CMM):

$$\frac{1}{8}zM \text{ bytes} \tag{16.1}$$

   ($z$ =no. of set bits in the query descriptor code).

2. The amount of data constituting all separator codes that could be represented in the output:

   This depends upon being able to specify the number of true matches, $A$. If there are $AM$-bit patterns, then the likely number of bits not set will be $(\frac{M-N}{M})^A$; set bits will therefore number $(1-(\frac{M-N}{M})^A)M$. Subtracting $(N-1)$ gives the number of separator code buckets (see Section 16.4.2.2). Assuming a 100% 1-bit error rate, each separator code bucket will contain an average of $\text{storage}_{1.0}/M$ separator codes,

**Table** 16.1: This table gives the amount of data to be retrieved for various values of $A$ and $z$. Having chosen $A$ and $z$, read off the two associated amounts of data, then sum to give the total amount of data which a CMM database index would require to answer a particular partial match query.

| A | 10 | 20 | 30 | 40 | 50 | 100 | 1000 |
|---|---|---|---|---|---|---|---|
| KB | 160 | 320 | 480 | 640 | 800 | 1530 | 9630 |
| $z$ | 2 | 4 | 6 | 8 | 10 | 12 | 14 |
| KB | 3500 | 880 | 230 | 67 | 29 | 21 | 21 |

each requiring $2N$ bytes storage[2]. Note that separator code buckets will not be uniform in size (although it is possible to make them uniform). It is therefore necessary to have some form of look-up table.

Error bits in the output also occur due to interactions in the CMM. These number $(M-N)q^z$, where $q$ is the proportion of weights set in the CMM (see [19]) and $\text{storage}_{1.0} \Leftrightarrow (q = 0.5)$. In summary:

$$\frac{\text{storage}_{1.0}}{M} 2N \left[ (1 - \left(\frac{M-N}{M}\right)^A) M - (N-1) + (M-N)q^z \right] \quad \text{bytes} \tag{16.2}$$

The total amount of data ((1) + (2)), given $M$ and $q$, depends only upon $A$ and $z$. To enable a comparison between the two methods, Table 16.1 summarizes CMM performance on an example case given in S&R, the example case being:

Total number of records = $2^{19}$
Number of segments = 64
Number of records per segment = 8192
Number of error bits at the segment level = 1/4 (per retrieval)
Number of error bits at the descriptor level = 1 (per retrieval)
Disk Page Size = 1 KB

From Table 16.1, it can be seen that with a medium-highly specified query (*i.e.* $6 < z < 14$), the amount of data is closely related to $A$, the number of true

[2] $2M$ bytes to allow up to $3 \times 10^6$ records, *i.e.* 16-bits to encode the position of each set bit.

matches (our desirable property). In comparison with SIC, CMM performs well: the worst case performance of SIC (when every segment is examined) is bettered by CMM (4MB for CMM *vs.* 10MB for SIC); in the unlikely event that only one segment descriptor matches, SIC does do better (CMM 800KB *vs.* SIC 80KB). The storage that would be required to implement this CMM database index is 28MB, compared with 15MB for Two Level SIC (as the size of the database increases, however, the relative storage for the two methods becomes less disparate, *e.g.* 1.6GB *vs.* 1.4GB for $2^{24}$ records).

### 16.4.3 Implementing a Predicate Calculus

Thresholding the output of the first CMM layer, coupled with the appropriate use of arity- and occurrence CMM units in this layer, can achieve a powerful reasoning behaviour. Both L-max and Willshaw thresholds (see Section 16.3.7) are used simultaneously: first, column sums are selected using an L-max criterion; next, a Willshaw threshold is applied to the selected column sums. The L-max criterion would normally be chosen to be the same as the number of bits in a separator token, but it may be the case that there are many more high column sums than expected. This would suggest that the output contained overlapped separator tokens (and it would then be possible to check a list of separator tokens and pass each in turn through the second layer).

#### 16.4.3.1 AND, OR and partial-AND

The level of Willshaw thresholding allows any of the operators **AND**, **OR** and **partial-AND** (see below) to be enforced in rule retrieval. For example, if the system has learned two rules:

1. **A** $\wedge$ **B** $\Rightarrow$ **i** $\Rightarrow$ **D**.
2. **A** $\wedge$ **B** $\wedge$ **C** $\Rightarrow$ **j** $\Rightarrow$ **E**.

And the tokens are as follows:

| | |
|---|---|
| A | 01100000 |
| B | 00000110 |
| C | 00011000 |
| i | 01000100 |
| j | 00100010 |

The superimposed input to the first CMM layer for rule 1 would be 01100110, and the rule would be trained in the arity 2 CMM unit (because there are two variables in the antecedent of the rule). For rule 2, the input

would be 01111110, and the rule trained in the arity 3 CMM unit. In retrieving an output from the first layer, 01111110 (6 set bits) would retrieve 00600060 from the arity 3 unit. If this is thresholded at 6 then the result is 00100010, the token for j. Thresholding at 6 therefore achieves an **AND**. Applying instead just 01100000 would retrieve 00200020, which gives 00000000, thresholding at 6; thresholding at 2, however, *does* result in the token for j. Using such a threshold is equivalent in this case to selecting any arity 3 rules that have A in one variable position, and this is an OR. Thresholding at an intermediate level (in this case 4) would allow inputs that represent any two of {A,B,C} to retrieve j, a function that we call partial-AND. Importantly, *all these functions are completed in a single pass through the system.*

#### 16.4.3.2 Rules with Disjunctions

For example:

$$A \vee (B \wedge C) \Rightarrow D.$$

To achieve *training* such a rule (as opposed to enforcing the OR function in *retrieval*), it is necessary to process the rule into two components before teaching:

1. $A \Rightarrow D.$
2. $B \wedge C \Rightarrow D.$

Each component is trained appropriately for for its new arity, the whole training process therefore possibly requiring as many passes as there are disjunctions. The application of 01100000 (which is only the token for A) to the arity 1 CMM unit, with appropriate thresholding, can now retrieve D (ignoring the separator stage for the sake of this example); similarly, applying 00011110 (our representation of $B \wedge C$) to the arity 2 unit also retrieves the correct output.

#### 16.4.3.3 Additional Features

The system can, in addition, support many other features that there is not space to detail here (such as negation and deletion, also detailed examples of dynamic variable binding).

## 16.5 Conclusions

CMM is a tool that has great potential for use in expert systems: at a basic level, it has very desirable storage characteristics, as well as some useful emergent properties (*e.g.* partial-AND can be evaluated in a single pass through the system); at a more sophisticated level, our CMM-based inference engine

supports predicate calculus. A realistic goal with the present system architecture, is to achieve massively parallel processing. In addition, the present system architecture is being extended to enable soft rule processing (by using the dimension of time in our representation).

# References


[1] Austin J. (1987) The Design and Application of Associative Memories for Scene Analysis. Unpublished doctoral dissertation, Brunel University.

[2] Austin J. (1995) Distributed Associative Memories for High Speed Symbolic Reasoning. Invited paper to the *International Journal on Fuzzy Sets and Systems.*

[3] Austin J. and Jackson T. (1994) The Representation of Knowledge and Rules in Hierarchical Neural Networks. In *Neural Networks for Knowledge Representation and Inference.* Hillsdale, N.J.: Lawrence Erlbaum Associates.

[4] Austin J. and Turner A. (1994) A Novel Method for Producing Near Orthogonal Codes. ACAG Internal Research Report 2734. University of York Computer Science Dept., York. YO1 5DD.

[5] Casasent D. and Telfer B. (1992) High Capacity Pattern Recognition and Associative Processors. *Neural Networks* (4) 5: 687–98.

[6] Filer R.J.H. (1994) ACAG Internal Research Report 2771. University of York Computer Science Dept.

[7] Gaschnig J. (1980) An Application of The PROSPECTOR System to The DOE's National Uranium Resource Evaluation. AAAI 1, 295–97.

[8] Knuth D.E. (1973) *The Art of Computer Programming* (3), Reading, Massachusetts: Addison Wesley.

[9] Lange T.E. and Dyer M.G. (1989) High Level Inferencing in a Connectionist Network. *Connection Science* (1) 2: 181–217.

[10] Lindsay R.K., Buchanan B.G., Feigenbaum E.A. and Lederberg J. (1980) *Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL Project.* New York: McGraw-Hill.

[11] Miller R.A., Pople H.E. and Myers J.D. (1982) INTERIST-I, an Experimental Computer-Based Diagnostic Consultant for General Internal Medicine. *N. Eng. J. Medicine* (307) 8: 468–76.

[12] Nadal J-P. and Toulouse G. (1990) Information Storage in Sparsely Coded Memory Networks. *Network* 1: 61–74.

[13] Rivest R.L. (1976) Partial Match Retrieval Algorithms. *SIAM J. Computing* 5: 19–50.

[14] Sacks-Davis R. and Ramamohanarao K. (1983) A Two Level Superimposed Coding Scheme for Partial Match Retrieval. *Information Systems* (8) 4: 273–80.

[15] Shastri L. and Ajjanagadde V. (1993) From Simple Associations to Systematic Reasoning. *Behavioural & Brain Sciences* (16) 3: 417–93.

[16] Smolensky P. (1990) Tensor Product Variable Binding and The Representation of Symbolic Structure in Connectionist Systems. *Artificial Intelligence* 46: 159–216.

[17] Sun R. (1994) Connectionist Models of Commonsense Reasoning. In *Neural Networks for Knowldege Representation and Inference*. Hillsdale, N.J.: Lawrence Erlbaum Associates.

[18] Touretsky D.S. and Hinton G.E. (1988) A Distributed Connectionist Production System. *Cognitive Science* 12: 423–466.

[19] Willshaw D.J., Buneman O.P. and Longuet-Higgins H.C. (1969) Non-Holographic Associative Memory. *Nature* 222: (June), 960–62.